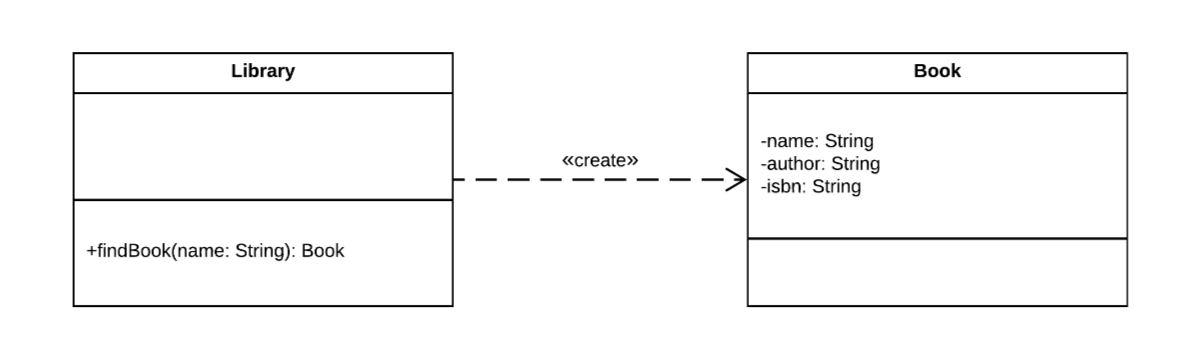
Relationships also have **multiplicity**, which shows how many instances of a class can exist on each side of a relationship.

**Dependency** relationships exist when classes depend on each other in such a way that a change to one class may affect the other, such as when one class accepts an instance of another class as parameter to a method.

* Dependency relationships are represented by arrows on dashed lines.  Stereotypes can be indicated within guillemets to provide further detail on the nature of the relationship.
* A dependency relationship can exist when we have a Library class that manages Book objects.  Since the Library class has a method that returns a Book, changes to the Book class could result in changes to the Library class (based on how Book objects are created)



public class Library {

public Book findBook(String name) {

//Do some book stuff.

return new Book();

}

}

class Book {

private String name;

private String author;}

private String isbn;

//Getters/setters omitted.

**Generalization**relationships exists when one class extends another class (making it a specialization of the parent class, like a Car is a specialization of a Vehicle).

* Generalization relationships are represented by a triangular arrow on a solid line.



* Operations and attributes on the parent class also exist on the child classes, without being explicitly specified.
* A generalization relationship can exist when we have a system that keeps track of vehicle rentals, where we have various specialized vehicles.

**Association** relationships often exist when classes have variables of other types, that they can invoke operations on.

* Association relationships are represented by an arrow on a solid line.
* Association relationships can be bi-directional, in which case both classes can reference each other.
* Association relationships can also include multiplicity, where we can have one instance on one side and exactly zero or one instance on the other side, or one instance on one side zero or more instances on the other side (**\*** refers to any number of instances), or any other combination.
* An association relationship can exist when we model the relationship between doctors and patients, where a doctor can have any number of patients and a patient can only be treated by one doctor at a time.A diagram of a patient

  Description automatically generated

public class Doctor {

  private String name;

private String registrationNumber;

private List<Patient> patients = new ArrayList<Patient>();

public void addPatient(Patient patient) {

this.patients.add(patient);

}

//Getters/setters omitted.

}

class Patient {

private String patientId;

private String name;

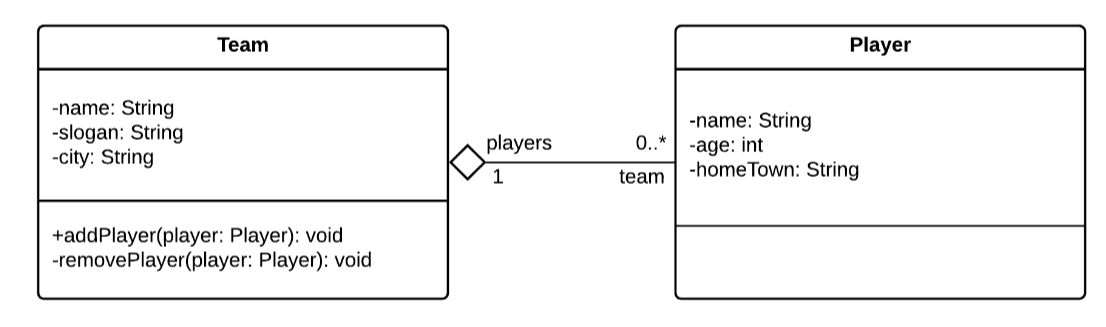
private int age;

//Getters/setters omitted.

}

**Aggregation**relationships exist when we aggregate (or bring together) objects of one class in another class.

* Aggregation relationships are represented by an unfilled diamond on the 'owning' side of the relationship.
* Objects on both sides of an aggregation relationship can exist in isolation.
* Aggregation relationships can have multiplicity.
* An aggregation relationship can exist when we model teams and players.  A player can exist without belonging to a team, and a team can exist without any players.



public class Team {

private String name;

private String slogan;

private String city;

  private List<Player> players = new ArrayList<>();

//Getters/setters omitted

public void addPlayer(Player player) {

this.players.add(player);

player.setTeam(this);

}

public void removePlayer(Player player) {

this.players.remove(player);

player.setTeam(null);

}

}

class Player {

private String name;

private int age;

private String homeTown;

  private Team team;

//Getters/setters omitted.

public void setTeam(Team team) {

this.team = team;

}

}

**Composition** relationships when objects are composed of (or made up of) other objects.

* Composition relationships are represented by a filled diamond on the 'owning' side.
* Objects in a composition relationship cannot, conceptually, exist in isolation.  This isn't always easy to enforce in code.
* If the parent object in a composition relationship is destroyed, so are the child objects.
* A composition relationship can exist when we model a system for creating web pages - pages cannot exist without a page header and a page body, and each PageHeader and PageBody object **must** belong to a WebPage object.

A diagram of a webpage

Description automatically generated

public class WebPage {

private final PageHeader header;

private final PageBody pageBody;

public WebPage(PageHeader header, PageBody pageBody) {

this.header = header;

this.pageBody = pageBody;

}

}

class PageHeader {

private String title;

private String charset;

//Getters/setters omitted.

}

class PageBody {

private String body;

//Getters/setters omitted}

**Enumerations**, or enums, are classes that provide a fixed set of literal values.

* Enums are indicated by the 'enumeration' keyword.
* Enums provide attributes (values), but generally do not provide behaviour, so they can be drawn without the operations section.
* Enums do not need to show attribute types, since all attributes are of the type of the enum itself.
* Enums also do not need to show access modifiers, since all attributes are implied to be accessible if the enum itself is accessible.

